An Aspect-Oriented Approach to Assessing Fault Tolerance

Jeffrey Cleveland, Joseph Loyall
Raytheon BBN Technologies
Cambridge, MA, USA
jeff.cleveland@alum.cs.umass.edu, jloyall@bbn.com

James Hanna
US Air Force Research Laboratory
Rome, NY, USA
james.hanna.4@us.af.mil

Abstract—**Fault tolerance and survivability are important aspects of many business-critical and mission-critical systems but it is still difficult to assess how well fault tolerance techniques work. Ensuring fault tolerance in military communication systems is particularly important due to the inevitability of hardware failure, data corruption, or service interruption and the risk that cascading failures could jeopardize critical military operations. In this paper, we present a fault tolerance assessment framework designed for distributed systems that provides automated injection of faults without changes to client or server code and automated assessment of whether the injected faults are tolerated. The framework applies aspect-oriented programming, specifically AspectJ, to inject faults and weave in assessment criteria. The framework supports both assessing the tolerance of direct faults, such as crashes and corruption, like traditional fault injectors, and conditional faults, which can be probabilistically, randomly, or periodically injected at runtime. This latter class of faults is not historically supported by fault injectors, but enables the assessment of tolerance to many important classes of faults threatening modern distributed military communication systems, including timing faults, resource exhaustion (e.g., denial-of-service), and integrity faults that are traditionally difficult to tolerate and assess. Additionally, the framework provides a centralized view for users enabling them to monitor and script coordinated tests comprising performance metrics and injected faults spanning services, applications, and hosts.**

Keywords—fault tolerance; assessment; testing; aspect-oriented programming; survivability

I. INTRODUCTION

To be useful in mission-critical or safety-critical military situations, modern software systems, such as those built around Service-Oriented Architecture (SOA) or publish-subscribe (pub-sub) paradigms, need to be tolerant of faults (whether due to malicious intent or not). Fault tolerance techniques have typically fallen into one or more of the following categories:

- *Fault Prevention*, which includes analysis and correction of problems in software and hardware before they lead to faults, as well as dynamic techniques such as restarting or reorganizing at regular intervals (e.g., based on observed Mean Time To Failure).
- *Fault Masking* includes tolerating, preventing, or neutralizing the effects of a failure. This often includes redundancy so that a failure in one part of a system is compensated for by redundant functionality in another part of the system.
- *Recovery*, which includes failing over to a backup, restarting functionality, or otherwise reacting to failure detection with corrective action.

Among the traditional masking techniques are replication-based approaches, which maintain copies of processes or services so that as long as one replica is alive, the service is available. Replication-based fault tolerance can consist of *active* or *passive* techniques, which differ in the way consistency in the state of replicas is handled.

Despite the importance of fault tolerance and the techniques available to provide it, assessing the fault tolerance and survivability of a system is still challenging for several reasons. First, fault tolerance is often built into a system in a special purpose manner (i.e., it is not an off-the-shelf component that can be purchased or acquired). Second, different fault tolerant techniques are more or less effective in particular parts of a system and in particular deployments. Finally, fault tolerance introduces a runtime cost, in terms of extra time (latency) and resources used (memory, bandwidth, and processing). This cost can negatively affect the performance of a system disproportionately to the benefit of the fault tolerance, if not designed and implemented appropriately.

In this paper, we describe an approach to assessing a system-under-test's susceptibility to faults and ability to handle faults. Our framework enables the injection of direct faults that take effect immediately in a running system and conditional faults that can occur either periodically, probabilistically, or randomly during operation of a system when a condition becomes true. Our prototype assessment framework supports the injection of faults into software without modifying the software and the automated detection of whether the fault is handled or not, utilizing the following:

- An *aspect-oriented programming (AOP)* [13] approach to specifying and weaving faults into existing code, without requiring developer-provided code changes.
- An *assertion feature* for specifying and evaluating whether an injected fault occurs or is handled by the system being assessed.

This work was supported by the U.S. Air Force Research Laboratory under Contract Number FA8750-10-C-0247.

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **OCT 2014**	2. REPORT TYPE	3. DATES COVERED **00-00-2014 to 00-00-2014**

4. TITLE AND SUBTITLE **An Aspect-Oriented Approach to Assessing Fault Tolerance**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Raytheon BBN Technologies,10 Moulton Street,Cambridge,MA,02138**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
Military Communications Conference (MILCOM 2014), Baltimore, MD, October 6 - 8, 2014.

14. ABSTRACT
Fault tolerance and survivability are important aspects of many business-critical and mission-critical systems but it is still difficult to assess how well fault tolerance techniques work. Ensuring fault tolerance in military communication systems is particularly important due to the inevitability of hardware failure, data corruption, or service interruption and the risk that cascading failures could jeopardize critical military operations. In this paper, we present a fault tolerance assessment framework designed for distributed systems that provides automated injection of faults without changes to client or server code and automated assessment of whether the injected faults are tolerated. The framework applies aspect-oriented programming specifically AspectJ, to inject faults and weave in assessment criteria. The framework supports both assessing the tolerance of direct faults, such as crashes and corruption, like traditional fault injectors, and conditional faults, which can be probabilistically, randomly, or periodically injected at runtime. This latter class of faults is not historically supported by fault injectors, but enables the assessment of tolerance to many important classes of faults threatening modern distributed military communication systems, including timing faults resource exhaustion (e.g., denial-of-service), and integrity faults that are traditionally difficult to tolerate and assess. Additionally the framework provides a centralized view for users enabling them to monitor and script coordinated tests comprising performance metrics and injected faults spanning services applications, and hosts.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**8**	

We use an AOP approach to assessing fault tolerance because faults will often manifest as effects on a specific cross-cutting concern and because AOP's weaving feature enables assessment of faults in places throughout the assessed system. Our approach targets software fault tolerance and targets a wide range of faults and assessment techniques. Other techniques typically target only direct or probabilistic faults, not conditional faults. Our prototype also provides support for establishing criteria to automatically assess tolerance.

The rest of the paper describes our prototype assessment framework and its application to an existing publish-subscribe information middleware system. We begin, in Section II, by describing aspect-oriented programming and its use in assessing fault tolerance. Section III describes our approach to automating assessment of fault tolerance. Section IV describes the design and implementation of our current prototype. Section V describes our application of the framework to the assessment of a real system. Section VI describes some related work. Finally, Section VII provides some concluding remarks.

II. Using AOP for Fault Tolerance Assessment

Aspect-Oriented Programming is a programming approach that enables the specification of cross-cutting aspects in high-level languages that get woven into source code at compilation or run time. In other words, while traditional languages, such as Java, support constructing a program around a dominant decomposition, such as the functional classes making up an application's business logic, aspects enable the separate programming of other aspects of a program's behavior – such as logging, synchronization, or quality of service (QoS) – that would have to be sprinkled throughout the functional classes.

An aspect weaver that works as a separate step of the compilation process automatically inserts the aspect code in the appropriate places throughout the functional application.

A. Overview of AOP

AOP offers an elegant approach to injecting faults and assertions into existing code in all the places that fault tolerance needs to be tested, without changing the original code of the program. We utilized AspectJ, an aspect-oriented extension to Java, which provides the following [12]:

- *Upward compatibility* – All legal Java programs are legal AspectJ programs.
- *Platform compatibility* – All legal AspectJ programs run on standard Java virtual machines.
- *Tool compatibility* – AspectJ works as an extension to existing tools, including integrated development environments (IDEs), documentation tools, and design tools.
- *Programmer compatibility* – Programming with AspectJ feels like a natural extension of programming with Java.

The central concept in AspectJ is that of a *join point*, a well-defined point in the program's flow of execution, e.g., a method call or variable access. AspectJ adds four new syntactic constructs to Java that affect a program's behavior at specific join points:

- A *point cut* specifies a join point programmatically. For example, a point cut can specify a method entry point and its arguments, which syntactically represent the join points associated with everywhere in the program's execution in which the method is called.
- *Advice* specifies code that is woven into a point cut and is executed when a join point is reached. The advice can be specified to be *before*, *after*, or *around*, which specifies whether the advice is executed prior to the ordinary functionality at the join point, after, or in place of, respectively.
- An *inter-type declaration* allows the programmer to modify a program's static structure, namely, the members of its classes and the relationship between classes.
- An *aspect* encapsulates the new constructs, similar to a *class* encapsulating functional code.

B. Advantages of Using AOP for Fault Tolerance Assessment

There are several advantages that we gained by using an AOP approach to specifying and executing the fault injection and assertions associated with our assessment framework. First, using AOP, and AspectJ in particular, enables the rapid development of a library of faults for testing a system and injection of those faults into an existing system. Because faults often manifest as effects on a specific cross-cutting concern, e.g., file-system calls from any object will fail if a disk fails, aspect-oriented programming is a natural paradigm to utilize for the fault injectors.

Second, we can weave the different kinds of faults needed to fully assess fault tolerance into the different places in which they can occur in a system. Currently, the framework supports the following types of faults:

- *Direct faults* that take immediate effect when triggered.
- *Conditional faults* which, when triggered, can take effect sometime in the future, repeatedly, probabilistically, and randomly.

Third, using AOP means that the framework is extensible to add new fault types, assertions, and injection points. The current prototype supports injecting faults into a service or its methods, a container, third party libraries, OS calls, a JVM, or a network connection.

Fourth, metric collection and logging are canonical use cases for AOP. Being able to observe system behavior in response to fault injection is an integral part of assessing fault tolerance.

Fifth, using AOP maintains a clear separation between the production and the test code. The code to assess the fault tolerance can be readily removed from the code that is deployed into target environments simply by not weaving in the aspects. This is a major advantage of being able to evaluate the fault tolerance of a system under development while eliminating the impact on the system's runtime performance.

Finally, support for AOP is readily available through an AspectJ library for Java and AspectJ plugins for build systems such as Maven and IDEs such as Eclipse.

III. AUTOMATING FAULT TOLERANCE ASSESSMENT

The main goal of our prototype framework is to automate the Fault Tolerance Assessment process, which includes the following three key aspects:

- A means to automate the injection of faults.
- A means to automate the analysis of results.
- Support for maintaining a library of faults that serve as tests.

With these three features, the assessment framework can be used in a continuous-build like manner (described in more detail in Section V.A). Our framework includes the notion of an *Experiment Template*, which is used to document a fault injection test and its assessment criteria, in a format that can be stored, reused, and modified. The following sections describe our approach to automating analysis and how we define the behavior of a test and its analysis criteria in an Experiment Template.

A. Assessing Tolerance to Injected Faults

To assess tolerance to injected faults, we measure system utility at certain endpoints. The current prototype measures utility at client endpoints, i.e., when messages are sent and received, and includes metrics such as message throughput and latency. Target values for these utility metrics should be based on expected system use case. When a fault results in a system falling outside of the acceptable range, the system can be seen as not tolerating the fault. Fig. 1 shows an example in which an injected fault manifests itself as an observable increase in message latency.

An alternative approach, which we did not take, would be to tie metrics to individual survivability techniques. For example, consider assessing fault tolerance based on measuring how long a service is down. This may accurately assess the effectiveness of a monitor-and-restart approach (faster recovery is better), but can be less useful to evaluate the fault tolerance of a system using an active replication approach. In this situation, the failure of a service (and it staying down) is completely masked by another service (a replica) taking its place, but not captured in a metric that simply measures how long the failed service is down. The success of masking or restart is measured by metrics looking for continued servicing of the service's clients.

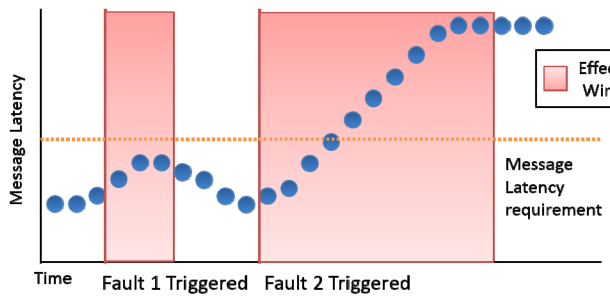


Fig. 1. Determining fault injection outcome.

Furthermore, tight coupling between survivability techniques and assessment results in additional developer effort for every technique implemented, i.e., developers have to add new assessment behaviors every time the fault tolerant system is modified. In our framework, assessment criteria are tied to the continued operation of the system in the face of faults, rather than to the particular techniques being employed to achieve continued operation.

When a fault is injected, we expect it to manifest itself within a certain time frame, i.e., the effect window. If system behavior moves out of tolerance range during that window we can say the system did not tolerate the fault.

B. Experiment Template

An Experiment Template defines the behavior of a test. This includes a script describing the faults that are injected, when they are injected, and a set of criteria that define if a test passes or fails.

A Script consists of Actions, each of which consists of a Fault Name, Fault Target, Fault Delay, and in the case of a Conditional Fault, the condition that triggers the fault. The Fault Name identifies the fault that will be injected. The Fault Target is the name of the service, component, or other system element into which the fault should be injected.

The Fault Delay is the period of time after a fault is injected that the system waits before injecting the next fault. If a criterion fails during this time frame, the most recently triggered fault is identified as the cause.

Criteria define what is considered acceptable behavior during a test, e.g., message throughput should not drop below a given threshold. This includes an "On Fail" status which determines what a criterion reports if its specified threshold is crossed. This can either be a *Warning*, which will be reported and the experiment continues, or a *Failure*, which ends the experiment. We also include a *Timeout* specification, in milliseconds, which represents the amount of time a criterion will wait between receiving metric events before expiring and failing. If system behavior is acceptable at the end of the effect window, the framework can trigger the next fault.

The execution of an Experiment Template is an *Experiment Instance*. The framework can launch multiple tests (Experiments consisting of multiple Experiment Instances) and log the results, to assess a system's overall fault tolerance.

IV. DESIGN AND IMPLEMENTATION

This section describes the design and implementation of our prototype assessment framework, including the mechanisms for triggering faults, monitoring system behavior, assessing system behavior, and user interaction.

A. *Fault Orchestrator and Controller*

A *Fault Orchestrator* coordinates the injection of faults during a controlled experiment. It uses *Fault Controllers* associated with the places in which faults can be injected, under direction of the Fault Orchestrator, as shown in Fig. 2.

The Fault Controller serves as an adapter between Fault Triggers and the Fault Orchestrator. These are added to target classes, e.g., a high level service, using inter-type declaration, and are initialized by a point cut and advice targeting the service's constructor. The *@DirectFault* and *@FaultCondition* annotations (described in Section IV.C) allow the fault controller to identify methods and fields to be identified at runtime using reflection. The fault controller is then able to directly trigger or modify the conditions under which the faults are triggered. The Fault Controller registers a callback to itself and the set of faults to which it has access with the Fault Orchestrator.

The Fault Orchestrator is a centralized view of all the distributed fault controllers and available faults within the system. It provides a control interface that is used by our web-based fault scripting tools to coordinate full system tests (described in Section IV.D).

B. *Event Collection*

To enable assessing system behavior, metrics and other system events are reported to a centralized location. We create the reporting mechanisms using classic AOP techniques for logging and metric collection and define several formats for these reports to ease analysis. These types are:

- *Metrics* report monitored system behavior, e.g., throughput at a client.
- A *Partial Metric* reports a piece of system behavior that needs to be combined with other partial metrics to be reasoned about. For example, the publisher of a message may report a partial metric containing an UID and send time while a consumer may report a partial metric containing a UID and arrival time. By combining these partial metrics we are able to calculate message loss or latency metrics.
- A *Trigger* indicates a fault was triggered and includes a fault name, target, and time.
- A *Report* is a notice of a system action that provides additional context to the user, e.g., a report may be sent when certain types of exceptions are triggered.

These reports are transmitted as HTTP POST requests to enable compatibility with not only our AOP-based metric collection, but also compatibility with other tools that can be used to monitor system behavior.

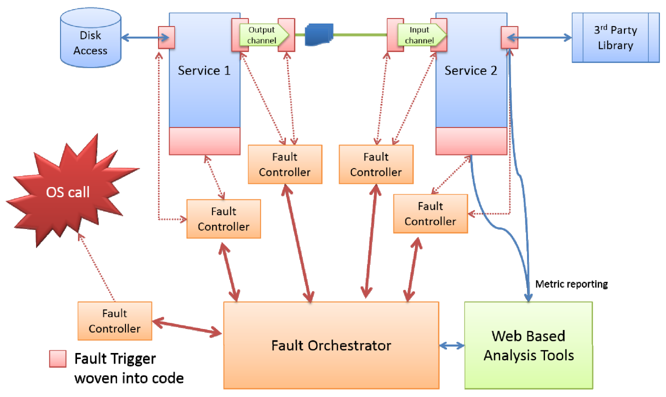


Fig. 2. Assessment system architecture.

C. *Fault Triggers*

Fault Triggers are pieces of code that enable a fault to be remotely injected. This includes both the trigger interface as well as the fault's logic. Our framework includes triggers for direct faults and conditional faults.

Direct faults are implemented as public methods added to the target class and inject the fault as soon as they are executed. AspectJ's support for inter-type declaration allows us to weave these methods into the code after compilation. The methods are annotated with *@DirectFault* which specifies a fault name and description. By calling a method annotated with *@DirectFault* the described fault will be injected. Fig. 3 shows a direct fault that gracefully stops a service when it is injected. Notice that the injected fault represents the effect regardless of the failure that led to the fault (e.g., either an external hardware failure or an internal software error).

```
@DirectFault(name="Stop service",
description="Calls stop()")
public void BaseService.stopService(){
        this.stop();
}
```

Fig. 3. Example of a Direct Fault.

Conditional Faults inject faults that take effect at a later time based on whether a condition becomes true. Conditional Faults have three components:

- A field of type *FaultCondition* is added to the target class using inter-type declaration and is annotated with *@ConditionalFault* (which includes the fault name and description).
- A point cut which describes where in the program execution the fault should be injected.
- Advice which is triggered when the join point described by the point cut is reached.

The advice checks if the FaultCondition returns true. If it does, then the specified fault code is triggered. If the

FaultCondition does not return true, the program execution proceeds as normal. By modifying the parameters in which a FaultCondition will return true, a conditional fault can be enabled or disabled. The current prototype supports the following built-in fault conditions:

- *DisabledFaultCondition*, which always returns false.
- *EnabledFaultCondition*, which always returns true.
- *ProbabilisticFaultCondition*, which returns true a (configurable) percentage of the time.
- *QuantityFaultCondition*, which returns true the first (configurable) n times.

The code in Fig. 4 adds a condition to an *InputChannel* class and a fault which adds one second of latency to reads from that input channel.

```
@ConditionalFault(name="Add latency",
   description="Sleeps for 1 second before
reading from a channel")
public LockableCondition
  InputChannel.__fault002 = new
LockableCondition();
before(InputChannel pic) :
  execution(* InputChannel+.read(..)) &&
target(pic){
    if( pic.__fault002.isTrue()) {
      Thread.sleep(1000);
    }
}
```

Fig. 4. Example of a Conditional Fault.

D. Web Based Analysis Tools

Our prototype includes a Tomcat-based web application with the following:

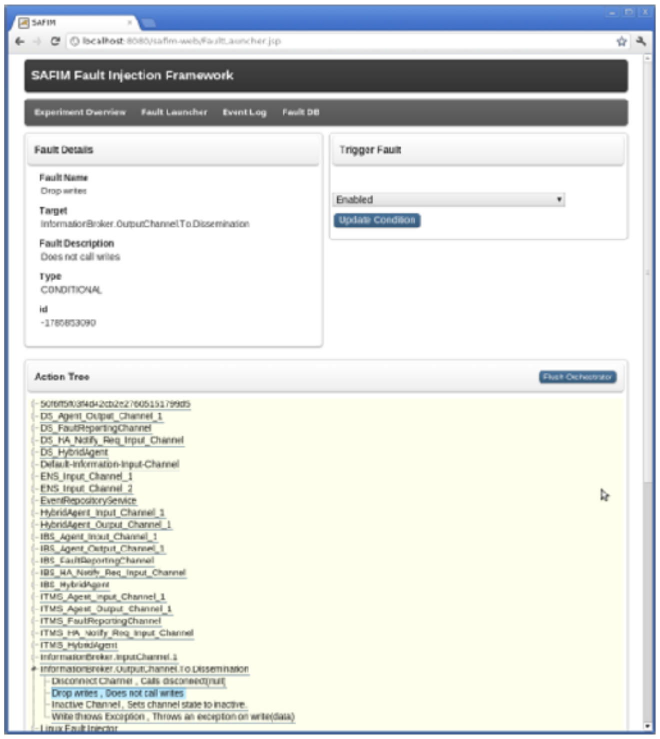
1. User interface,
2. Metric collection,
3. Executor for the experiment template,
4. Analysis tools.

The Web-based User Interface (UI) assists developers trying to assess system survivability. It includes a *Fault Launcher window* (Fig. 5(a)) that displays all of the currently registered faults and an interface that allows a user to manually trigger and enable specific faults.

The UI also includes an Event Log which contains a time-based graph of all reports, events, and metrics received by the framework and a detailed log of events (Fig. 5(b)).

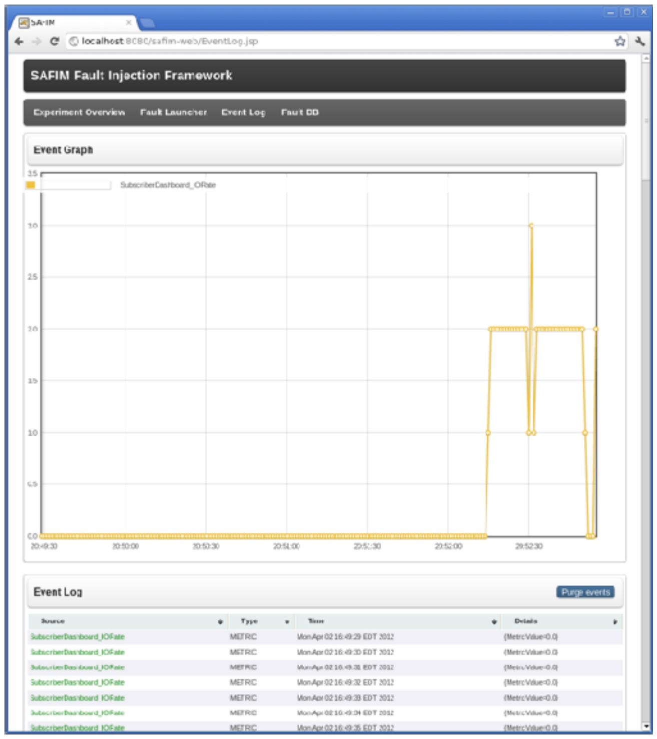
The prototype includes a *Fault Model Database* that includes all the faults and targets that have been registered with the Fault Orchestrator. These are accessible through a UI that enables viewing and updating the contents of the database. The contents of the database are used to auto-complete fields while creating a new Fault Script.

The web-based UI also enables a user to display all of the Experiment Templates; to execute, delete, or edit templates; or to create new experiment templates. Another page of the web-based UI allows the details of a given experiment template to be displayed, along with a detailed history of executions of the experiment, the template's actions, and its criteria. This page



Fault Name
Drop writes
Target
InformationBroker.OutputChannel.To.Dissemination
Fault Description
Does not call writes
Type
CONDITIONAL
id
-1785853090

```
50f6ff5f03f4d42cb2e27605151799d5
DS_Agent_Output_Channel_1
DS_FaultReportingChannel
DS_HA_Notify_Req_Input_Channel
DS_HybridAgent
Default-Information-Input-Channel
ENS_Input_Channel_1
ENS_Input_Channel_2
EventRepositoryService
HybridAgent_Input_Channel_1
HybridAgent_Output_Channel_1
IBS_Agent_Input_Channel_1
IBS_Agent_Output_Channel_1
IBS_FaultReportingChannel
IBS_HA_Notify_Req_Input_Channel
IBS_HybridAgent
ITMS_Agent_Input_Channel_1
ITMS_Agent_Output_Channel_1
ITMS_FaultReportingChannel
ITMS_HA_Notify_Req_Input_Channel
ITMS_HybridAgent
InformationBroker.InputChannel.1
InformationBroker.OutputChannel.To.Dissemination
  Disconnect Channel , Calls disconnect(null)
  Drop writes , Does not call writes
  Inactive Channel , Sets channel state to inactive.
  Write throws Exception , Throws an exception on write(data)
Linux Fault Injector
```



Source	Type	Time	Details
SubscriberDashboard_IORate	METRIC	Mon Apr 02 16:49:29 EDT 2012	{MetricValue=0.0}
SubscriberDashboard_IORate	METRIC	Mon Apr 02 16:49:30 EDT 2012	{MetricValue=0.0}
SubscriberDashboard_IORate	METRIC	Mon Apr 02 16:49:31 EDT 2012	{MetricValue=0.0}
SubscriberDashboard_IORate	METRIC	Mon Apr 02 16:49:32 EDT 2012	{MetricValue=0.0}
SubscriberDashboard_IORate	METRIC	Mon Apr 02 16:49:33 EDT 2012	{MetricValue=0.0}
SubscriberDashboard_IORate	METRIC	Mon Apr 02 16:49:34 EDT 2012	{MetricValue=0.0}
SubscriberDashboard_IORate	METRIC	Mon Apr 02 16:49:35 EDT 2012	{MetricValue=0.0}

Fig. 5. A Web interface that enables the user to (a) manually inject faults (left) and (b) observe system behavior (right).

also provides the ability to add new actions and criteria to the template.

The history of experiment executions includes the following details:

- Whether a Warning was recorded.
- Whether a Failure occurred (i.e., an injected fault was not tolerated).
- Start and end times for instances of experiment executions.
- An event graph and table for events associated with specific experiment instances.
- A summary of the faults that were injected during experiment executions.
- A summary of criteria that were evaluated, i.e., to evaluate whether a fault was tolerated.

V. APPLICATION OF THE ASSESSMENT FRAMEWORK TO A PUB-SUB SYSTEM

This section provides a qualitative evaluation of implementing and assessing fault injection using our framework. We performed our evaluation based on a fault model that we developed for an existing service-oriented publish-subscribe information broker [7]. The fault model defines specific and detailed software faults that can manifest themselves in all components of the pub-sub software, but that fall into the five broad categories shown in Fig. 6(a). Based on this detailed fault model, we derived a set of detailed faults that could be injected by the framework, which fall into the broad categories shown in Fig. 6 (b).

We used the framework to implement faults in each of these and at the different localities in which they could manifest themselves (e.g., in system service logic, containers, third-party code, etc.). In all cases, we found our framework able to support the implementation of faults from within our fault model, as well as metrics and criteria capable of reflecting the usage requirements of our target application. The following sections provide some details of the specific faults and criteria at the level a developer hoping to utilize our framework for a new application would need.

A. Mapping Fault Categories to AOP based Injection

We implemented faults from each of the following categories: crash faults, timing faults, value faults, and omission faults (we have not yet addressed Byzantine faults). In addition, we implemented faults with varying localities, including at the service level, at the process level, and at the node level.

We implemented crash faults at the service, container, and node level. Each case was implemented as a direct fault. At the service level, the aspect code calls specific pieces of the service's *stop* method. To cause a crash at the JVM level, the aspect code calls *System.exit*. To cause a crash at the OS level, a fault controller that runs with root privilege invokes a direct fault that makes a system call to shutdown the OS.

We implemented the timing faults with conditional faults affecting channels, a communication layer abstraction provided by the pub-sub software. The conditional faults add a sleep (periodically, probabilistically, or a certain number of times) to write and read calls.

We implemented value faults at locations that the detailed fault model described as susceptible to value faults, e.g., due to incorrect inputs or errors in algorithms. We defined direct faults that randomize and nullify the state of various services. We also defined conditional faults that scramble data being received over network connections.

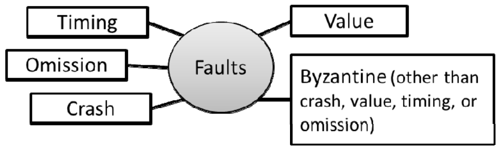
We also defined omission faults at several locations. We defined omission faults that emulate disk failures and result in IO exceptions being thrown whenever attempts to read or write certain file system paths were made. Finally, we defined faults that simply drop information being read from or written to a network resource.

As mentioned in Section III, our assessment framework prototype is being used as part of a nightly build process for the pub-sub system. The software is built and then an automated test framework runs through the full set of faults covering the fault model, using our prototype to inject each and assess whether the system tolerates the fault.

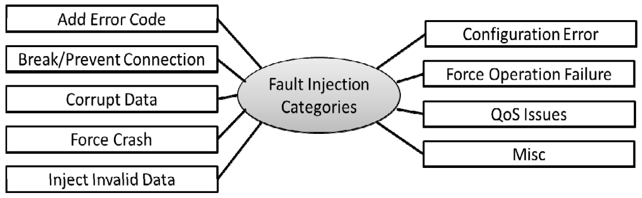
B. Metric Collection and Criteria

For our evaluation, we monitored the assessment criteria at the publication and subscription clients. At these endpoints, we were able to instrument end-to-end latency, loss of information, and throughput. These metrics were sufficient for us to assess the tolerance (or not) of crash, timing, and omission faults. In the case of value faults, we designed the experiments so that the value faults, if not tolerated, would affect the latency, loss, or throughput, so that we could utilize the same metrics.

Latency and loss were both reported at each endpoint as partial metrics. A piece of aspect advice inserts a UID in each published message, and then publishes that UID and the current time. When the subscriber receives the message it publishes a partial metric containing the UID and the current



(a) Categories of software faults in a pub-sub information broker



(b) Categories of injected faults derived from the fault model.

Fig. 6. Fault model and categories of faults used to evaluate the assessment framework.

time. The framework uses these partial metrics to calculate latency and to identify messages that are not received within a specific time (the effect window) as being lost. To collect and report throughput, the subscribing client simply reports the number of messages it receives per second.

While we have described just a few of the metrics and criteria that we used for fault tolerance assessment, our use of AOP for our approach enables a much wider range.

VI. Related Work

A. Fault Injection

There are several fault injection frameworks available including research and commercial tools. Many of these inject at lower levels, including Ferrari [10], which utilizes software traps associated with memory accesses or timeouts, and ORCHESTRA [4], which utilizes network level fault injection.

Our approach is targeted more toward the software fault tolerance associated with software services and the failures that might occur in software logic, corruption, misconfiguration, and so forth. The Hadoop File System [8] includes a fault injection framework built using AspectJ similar to that which we describe in this paper. The main differences between our framework and Hadoop fault injectors is that the Hadoop fault injector only supports probabilistic faults, i.e., not direct or other conditional faults, and our framework provides richer support for establishing criteria and metrics, and user interfaces supporting the assessment of fault tolerance.

Chaos Monkey [2] is a fault injection framework recently released by Netflix, targeting services running in elastic cloud settings. The difference between our approach and Chaos Monkey is that Chaos Monkey only triggers crash faults and is not capable of covering the ecosystem of potential faults.

Byteman [5] is an advice injection tool for Java that has also been used for fault injection within Java applications. Byteman interacts with a Java application via a JVM agent capability and allows a user to interact with the system using Event Condition Action rules written in the Byteman scripting language. In contrast, our framework exploits the programmer compatibility of AspectJ, resulting in fault injection code that looks and feels like the Java code of the targeted application. Byteman's ECA rules include conditions that have to evaluate to true before the rule action is executed, so that it supports a form of conditional faults.

B. Fault Detection/Diagnosis

Research on fault detection and diagnosis in industrial settings [3] rely on quantitative and qualitative model and process history-based methods [17] and expert systems approaches [9].

C. Aspect Oriented Programming

AOP has been used for other purposes beyond the fault injection and assessment functionality that we are targeting in this paper. Although logging is used as the common example of AOP's use, it has grown into wide usage for many cross-cutting concerns, including access control and security [15][18], storage management [11], and QoS management and monitoring [6] in object- and component-oriented systems.

Techniques related to AOP have also emerged, although none appear to have gained the traction and approached the level of adoption. Two of the best known are Composition Filters [1], which utilizes wrapping and interception, and Subject-Oriented Programming or HyperSpaces [16], which eliminates the dominant decomposition (i.e., the functional concern) of AOP and treats everything as aspects that are composed to create an application.

VII. Conclusions

We have developed a prototype framework for assessing the survivability of distributed (and standalone) systems. Our framework utilizes AspectJ to provide not only a technical compatibility with target Java systems, but also programmer and tool compatibility. The use of AspectJ enables the collection of platform metrics and logging, which are canonical examples of AOP. We implemented this fault injection and metric collection within the context of a centralized Fault Orchestrator which provides users a direct interface to the assessment framework and a means of automating fault injection behaviors. We have used the framework to assess the fault tolerance of an existing distributed publish-subscribe middleware system.

There are several directions for this work to progress in the future. Immediate gain could be accomplished by integrating this work with existing build and development tools. For example, hooks between the Fault Orchestrator and an automated build and continuous integration system such as Jenkins or custom fault controllers would enable the triggering of faults from test suites such as JUnit.

Another opportunity for future work is further automating fault injection behaviors. In Pal et al [14], an automated means of determining the relationships between network events capable of compromising a running system is explored. Such techniques could be utilized to better assess system susceptibility to cascading faults.

References

[1] M. Aksit, B. Tekinerdogan, and L. Bergmans, "Achieving adaptability through separation and composition of concerns," Special Issues in Object-Oriented Programming. M. Muhlhauser (Ed.), (1996), 12-23.

[2] "Chaos Monkey released into the wild, http://techblog.netflix.com/-2012/07/chaos-monkey-released-into-wild.html.

[3] L. Chiang, E. Russell, and R. Braatz, Fault Detection and Diagnosis in Industrial Systems, Springer, 2001.

[4] S. Dawson, F. Jahanian, and T. Mitton, "ORCHESTRA: a probing and fault injection environment for testing protocol implementations," Proc. IEEE International Computer Performance and Dependability Symposium, Urbana-Champaign, Illinois, September 4-6, 1996.

[5] A. Dinn, "Flexible, dynamic injection of structured advice using Byteman," Proc. Tenth International Conference on Aspect-Oriented Software Development, Porto de Galinhas, Brazil, ACM, March 21-25, 2011, 41-50.

[6] G. Duzan, J. Loyall, R. Schantz, R. Shapiro, and J. Zinky, "Building adaptive distributed applications with middleware and aspects," Proc. International Conference on Aspect-Oriented Software Development (AOSD '04), Lancaster, UK, March 22-26, 2004.

[7] R. Grant, V. Combs, J. Hanna, B. Lipa, and J. Reilly, “Phoenix: SOA based information management services,” Proc. SPIE Defense Transformation and Net-Centric Systems Conference, Orlando, Florida, April 2009.

[8] “Hadoop fault injection, http://hadoop.apache.org/hdfs/docs/r0.21.0/-faultinject_framework.html

[9] J. House, W. Lee, and D. Shin, “Classification techniques for fault detection and diagnosis of an air-handling unit,” ASHRAE Transactions: Symposia (January 1999), 1067-1097.

[10] G. Kanawati, N. Kanawati, and J. Abraham, “FERRARI: a flexible software-based fault and error injection system,” IEEE Trans. on Computers, 44, 2 (1995), 248-260.

[11] D. Kaul, A. Gokhale, L. Dawson, A. Tackett, and K. McCauley, “Applying aspect oriented programming to distributed storage metadata management,” Proc. Workshop on Best Practices in Applying Aspect-Oriented Software Development (BPOAOSD), Vancouver, British Columbia, Canada, March 13, 2007.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, “An overview of AspectJ,” Proc. ECOOP 2001—Object-Oriented Programming (2001): 327-354.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin, “Aspect-oriented programming,” Proc. ECOOP'97-Object-Oriented Programming, 11th European Conference, LNCS 1241 (1997), 220-242.

[14] P. Pal, R. Schantz, A. Paulos, B. Benyo, D. Johnson, M. Hibler, and E. Eide, “A3: an environment for self-adaptive diagnosis and immunization of novel attacks,” Proc. Adaptive Host and Network Security Workshop, IEEE International Conference on Self Adaptive and Self Organizing Systems, Lyon, France, September 12-14, 2012.

[15] R. Sethi, “Aspect-oriented programming and security,” http://www.sym-antec.com/connect/articles/aspect-oriented-programming-and-security, November 2, 2010.

[16] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, Jr., “N degrees of separation: multi-dimensional separation of concerns,” Proc. International Conference on Software Engineering (ICSE'99), Los Angeles, CA, May, 1999.

[17] V. Venkatasubramanian, R. Rengaswamy, K. Yin, and S. Kavuri, “A review of process fault detection and diagnosis,” Computers and Chemical Engineering, 27, 3 (March 2003), 293-346.

[18] J. Viega, J. Bloch, and P. Chandra, “Applying aspect-oriented programming to security,” Cutter IT Journal, 14, 2 (February 2001), 31-39.